A minimal GDB stub for embedded remote debugging.

Minheng Tan
Columbia University
12/12/2002

Abstract

Debugging software running on an embedded chip is more difficult than doing the same on a personal computer with well-established user-friendly environment. The problem with embedded chips is that there is minimal support to facilitate any debugging process. Often found on PCs and not on the smaller scale embedded chips are operating system support, threading support, IO support, user interface support, etc. Without these, engineers simply cannot run a debugger on the embedded chip to test buggy software. To mitigate these deficiencies, engineers often separate the debugging setup by running small stub[1] software on the small embedded chip and leave the rest non-crucial and machine-independent debugger running on a more powerful remote machine.

This paper describes the implementation of a minimal GDB server that runs on an x86 processor. This demonstrates how to split a debugging session into two halves: a server that runs on the target being debugged and the debugger that runs on a host machine. The example server implements the minimum required number of commands. The goal is to run as little code as possible on the chip. Anything that the remote machine can take care of gets isolated and runs on a remote machine. Symbol information, for example, is non-essential to debugging an application so the remote machine instead of the stub can manage it. User interface is also a non-essential concern. The remote machine has the luxury of developing a grand user interface, but it makes less sense to have to run this code on the chip.

This paper provides insight as to how to work around a situation where there's limited computing power by leveraging computing power elsewhere. It focuses on what is needed while what can be exfoliated from the stub. This paper serves as a guideline for how to setup a debugging session with future and current embedded chip software development.

Introduction

There are many ways to test software running on an embedded chip. One way is to test the software by running it on a simulator for the embedded chip. A well-written simulator can also behaves like a debugger for that it can always stop a process by pausing the simulation. It can easily provide information on the variables, registers, or anything programmers wish to find out by looking up program memory that it already simulates. But a simulator can be costly to construct if one is not readily available.

Another feasible option considers remote debugging. The buggy software runs on the actual board instead of a simulator. A small stub program runs along side the tested program and instruments the tested program. This stub is responsible for stopping, continuing, reading, and writing registry/memory, etc. It also communicates with the full debugger that oversees and manages the entire debugging operation. The full debugger runs on a relatively powerful remote machine. It communicates and gives command to the stub, which then carries out and returns with the results. A debugging session consists of a sequence of request and reply between the debugger and the stub.

The most prominent command issued by the debugger is perhaps the breakpoint command, followed by run, by read memory addresses, and so forth. This correlates to a user at a terminal issuing a breakpoint on a line of source code, runs the program, wait until the program stops, and reads some memory addresses that corresponds some variables in the program while trying to understand the dynamic of what's going on under the hood. Thus with remote debugging achieves the net effect of debugging software running on the board, but without running the full debugger on the target board.

The question arises on what exactly does the stub do and how does the full debugger interact with the stub. Most importantly, what the stub cannot do without.

This paper provides an analysis. Briefly, the stub needs only be able to respond to a debugger's command to read write register memory and resume programs. The stub code behaves unintelligently. The full debugger must have knowledge of the target chip and be able to drive the stub intelligently. Analogously, an operating system drives a kernel driver to control a device. Here the debugger drives the stub by issuing debugging related commands. For example, the debugger wishes to set a break point at the certain line in the source code. It consults the symbol table and finds out the corresponding program address. It then issues a command to read the instruction stored at the address. Then it writes an interrupt instruction to that address location. The breakpoint is set. The debugger thus achieves the user's objective by working with the few commands handled by the stub.

Related work

Programmers, prior to the popularity of debuggers, insert printf statements into codes to gain insights on the dynamics of buggy software. Since the first introduction of debuggers, there are numerous debuggers and techniques introduced. The more common debuggers include cdb, msdev, dbx, windbg, kdb, gdb, etc….

Most debuggers work in similar fashion. They pause the execution of a running process based on some criterions and allow the user to examine the program internals while the program is paused. The common way to achieve this is to patch the program with special instructions so that program relinquishes control of the CPU when it executes the instructions. For example on a x86 machine the special instruction is byte "CC", which will cause the program to fault and relinquish control to a signal handler.

While runtime code patching is not new, Buck and Hollingswoth [1] describe an API for doing runtime code patch-

[1] Stub is referred to as the target, nub, etc in other literatures.

ing. They create common APIs at the machine independent level. The actual patching work is done separately using different modules. This approach enables portability, and most importantly resembles the remote debugging paradigm.

Then again, remote debugging is not something new. Hanson and Raghavachari [2] introduce a machine-independent debugger called *cdb*. The paper aims to separate the debugger and the stub, which is called a 'nub' in their paper. The nub is compiled with the program to be debugged. It communicates with the debugger to facilitate the debugging environment. This is the major design constituting remote debugging.

GDB is a more widely used debugger. It also provides remote debugging capabilities. The debugged stub is called a target. GDB is also extensible for that it can communicate with a variety of remote stubs to perform debugging tasks. A sample stub, called GDBServer, comes with the gdb. The debugger and the stub communicate using its own protocol called the Remote Serial Protocol.

The remote serial protocol is developed for gdb so that there is a standard protocol for stub writers to communicate with gdb. Gatliff [3] gives a summary the remote serial protocol. He also includes examples of the message exchanges used in the protocol. The protocol is basically a request and reply, which is very similar to the http protocol. The debugger encodes the request in an ASCII string, sent to the server (stub) and then waits for an ASCII string reply. The protocol is significantly simple so it can run on top of virtual all communicate medium.

GDB is so widely used that it is selected as the sample in this paper. Others find this debugger useful as well. Kawachiya and Moriyama [5] describe their adaptation of gdb to their own hardware, Engineering Support Processor. They ported the GDBServer stub to the architecture and describe the processing of doing so. Their paper also examines some aspects for rewriting the stub.

Debuggers that support remote debugging

Remote debugging is not new. Most sophisticated debuggers offer some support for debugging software remotely.

I choose gdb because it is extensible, open source and supports numerous platforms. The current gdb, version 5.2.1, supports Intel 386, Motorola 680x0, Hitachi SH, SPARC, and Fujitsu SPARCLITE. This means the debugger has knowledge of these platforms and knows how to debug software running on these platforms. The real gem is that the debugger does not necessarily need to run on these platforms, despite also having existing ports to these platforms. In short, gdb knows how to set/restore interrupt instructions, unwind stacks, and fiddle with the registers for all of these platforms. But by itself, gdb does not necessarily do these tasks. It requests a software stub designed for a particular platform to do the actual low level fiddling, such as reading writing to memory and registers. This approach makes gdb extremely extensible for all it takes for gdb to support a new version of the same platform is to rewrite the appropriate stub if need be.

Because of the architecture, debugging remote programs comes naturally to gdb. The only additional requirement to enable remote debugging is that the communication between the debugging stub and the debugger through some medium.

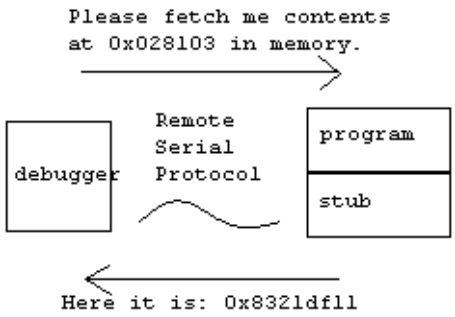

Figure 1. Debugger requesting the stub to read memory.

It's worth mentioning the communication between the stub and the debugger. GDB can use numerous communication protocols. e.g. TCP/IP, udp, or serial cable. The example GDBServer runs on top of TCP/IP, but any protocol would suffice as long as the stub and the debugger can convey the right information. The preferred way is often limited to the capability of the chip and is not a concern in this paper. Different communication protocol/medium does not change what are the essential things on the stub other than the communication portion of the code.

The gdb distribution comes with a sample stub call GDBServer. The stub runs as a standalone process on a remote machine, which also can be the same machine that gdb runs on. As an example, this stub does not impress because it requires the same operating system support as the debugger itself. This is to say that if GDBServer can run on a machine, then gdb can also run on the machine. Nonetheless, it serves as an example of what the debugger requests stubs do and how to implement these requests. A stub that works in similar fashion but without the operating system support can be easily derived from this example.

Example GDBServer

I created a sample gdbserver by following the same guidelines outlined for the original gdbserver. The stub runs on the x86 architecture. It runs as a separate user process on the Red Hat Linux operating system and can easily be ported to other x86 platforms. It implements a subset of the Remote Serial Protocol, while remaining capable to debug various programs running on the platform. It only answers on the TCP/IP protocol, whereas the real GDBServer can work on other protocols including udp and serial ports. Upon receiving a request, the stub translates the request into ptrace calls to the operating system. Then the stub returns the results from the ptrace calls to the debugger in a messages reply. Here's a list of commands the sample stub implements.

Command	In Stub	Description
g	yes	Reads all registers
G	yes	Writes all registers
m	yes	Reads memory at address
M	yes	Writes memory at address
?	yes	Get last signal: S
s	yes	Step the program
c	yes	Continues program execution

Table 1. List of commands implemented by example stub.

Because the stub runs as user process in the operating system, it does not have all of the privilege on a machine. So the stub depends on the operating system to do its low level fiddling. For example, to read from memory, the stub requests the operating system through a ptrace call with the PTRACE_PEEKTEXT argument. The operating system is responsible to read memory pages from the designated process memory address space, along with all the necessary page table lookup and translations. For stubs that run without the operating system, it needs to do the above task itself. To a read a register, the stub also consults the operating system through a ptrace call. The operating system in turn looks up the block that contains the saved registers of the preempted process and returns the appropriate data. Again, a stub that runs without the operating system needs to figure out the location of the saved registers and fetch the saved data. So operating system support is absolutely required for this example stub.

Here's how to stub implements the commands. The stub supports the "g" and "G" command, which are read and write register respectively. The results of the "g" command are a string of hex ASCII representing all of the registers in a ordering expected by gdb. Here's an excerpt of the g command.

```
buf = malloc (regset->size);
res = ptrace (PTRACE_GETREGS,
              childpid,
              0,
              (int) buf);
```

Figure 2. Excerpts of register read implementation.

The buf variable, after the ptrace call, points to a buffer storing the registers. The "G" command is the complement of the "g" command and is followed by string of hex ASCII representing all of the registers. Here's the call to write the registers.

```
regset->fill_function (buf);
res = ptrace (PTRACE_SETREGS,
              childpid,
              0,
              (int) buf);
```

Figure 3. Excerpts of register write implementation.

The above ptrace calls writes the data pointed by the buf variable to the saved registers block.

As for memory related commands, the stub supports the "m" and "M" command to read and write memory, respectively. The starting address and number of bytes follows directly after the "m" character in the packet. The response is a sequence of hex ASCII representing the memory content of the requested memory address location. Here's the helper function that reads the data in 'memaddr' address of the debugged process to 'myaddr' buffer.

```
void read_memory (long long memaddr,
                  char *myaddr,
                  int len) {
 int i;
 PTRACE_XFER_TYPE *buffer;
 long long startAddr = memaddr & -(long long)
                       sizeof(PTRACE_XFER_TYPE);
 long long endAddr = (memaddr + (long long)len);

 endAddr = endAddr & -(long long)
           sizeof(PTRACE_XFER_TYPE);

 buffer = (PTRACE_XFER_TYPE*)
          alloca( ((endAddr - startAddr)/4 + 1) *
          sizeof(PTRACE_XFER_TYPE) );

 i = 0;
 while (startAddr <= endAddr) {
  buffer[i] = ptrace(PTRACE_PEEKTEXT,
                     childpid,
                     (PTRACE_ARG3_TYPE) startAddr,
                     0);
  startAddr += sizeof(PTRACE_XFER_TYPE);
  i++;
 }

 memcpy(myaddr,
        buffer + (memaddr&
        (sizeof(PTRACE_XFER_TYPE)-1)),
        len);
}
```

Figure 4. Excerpts of the memory read implementation.

The complementary command "M" is accompanied by the starting memory address, size of the data, and the data in hex ASCII form. Here's the other helper function that writes the data in 'myaddr' to 'memaddr' address of the debugged program.

```
int write_inferior_memory (long long memaddr,
                           char *myaddr,
                           int len) {
 int i;
 int count;
 PTRACE_XFER_TYPE *buffer;
 long long startAddr = memaddr & -(long long)
                       sizeof(PTRACE_XFER_TYPE);
 long long endAddr = (memaddr + (long long)len);

 endAddr = endAddr & -(long long)
           sizeof(PTRACE_XFER_TYPE);
 count = (endAddr - startAddr)/4 + 1;

 buffer = (PTRACE_XFER_TYPE*)
          alloca( count * sizeof(PTRACE_XFER_TYPE) );

 buffer[0] = ptrace(PTRACE_PEEKTEXT,
                    childpid,
                    (PTRACE_ARG3_TYPE) startAddr,
                    0);
 if ( count > 1 ) {
  buffer[count-1] = ptrace(PTRACE_PEEKTEXT,
                    childpid,
                    (PTRACE_ARG3_TYPE)endAddr,
                    0 );
 }

 memcpy ((char *)buffer + (memaddr &
          (sizeof(PTRACE_XFER_TYPE)-1)),
```

```
            myaddr,
            len);

 i = 0;
 while ( startAddr <= endAddr ) {
  errno = 0;
  ptrace (PTRACE_POKETEXT,
           childpid,
          (PTRACE_ARG3_TYPE) startAddr,
          buffer[i]);
  if (errno) {
   return errno;
  }
  i++;
  startAddr += sizeof(PTRACE_XFER_TYPE);
 }
 return 0;
}
```
Figure 5. Excerpts of memory write implementation.

In addition, the stub responses to three control-related commands: get last signal, step, and continue command. The stub responses to the get last signal command by sending the reason that stops the debugged process. This information is given to the signal handler when the debugged program is interrupted, and is transferred from the operating system back to the stub. The mechanism to do so is through a wait on ptrace call. A stub without operating system support would have to save the signal information somewhere in its memory and transfer to gdb when asked to. This is true for any signal such as bus error, access violation, out of memory, etc. This information is transferred back to gdb, which then entails a sequence of register and memory read commands. Once gdb gathers enough information from the register and memory, it can compute and display to the user the current point of execution of the debugged program along with the last signal. Here's the code segment that captures the last signal in variable 'w'.

```
unsigned char waitForProcess (int childpid, char *status) {
 int pid;
 int w;

 pid = waitpid( childpid, &w, 0 );
 *status = 'S';
 return (unsigned char) WSTOPSIG(w);
}
```
Figure 6. Excerpts of get last signal implementation.

The step and continue command tells the stub to execute one instruction and continue execution until signaled, respectively. The stub calls ptrace with PTRACE_SINGLESTEP or PTRACE_CONT for stepping and continuing. A stub without the assistance of the operating system needs to manually restore the debugged process's context and switch[2] to it. Finally, here's the code to continue and step the debugged process, respectively.

```
 ptrace (PTRACE_CONT, childpid, 1, 0);
 …
 ptrace (PTRACE_SINGLESTEP, childpid, 1, 0);
```
Figure 7. Excerpts of the step and continue implementation.

The separated and the non-separable

Here I discuss the functionalities that are absolutely necessary to run on the stub and those that the full debugger performs.

Regardless of how much or little intelligence a stub has, it must be able to handle an exception or signal of the debugged program, for that is the time the debugged program unwillingly relinquish control of the CPU. The processor calls the signal handler either deliberately so on a user breakpoint or when the program encounters a fault. The handler needs to convey this fault information back to the debugger.

The sample stub does this using a ptrace wait command, which waits until the debugged program reaches the signal handler. By itself, it does not setup the signal handler for the debugged program as the operating system facilities this procedure. Nonetheless, any stub needs some method of intercepting and handling a signal from the debugged program.

Resuming the stopped program is also a responsibility of the stub. The stub must be able to restore the program context and switch to the program when requested by gdb. The example stub does this through the operating system, which already saves the context of the program. Resuming a stopped program in a multiprocessing operating system is as simple as fiddling around with the scheduler that already supports this kind of operations.

Fetching and writing the registers and memory are also another responsibility of the stub. A stub that runs with no operating system support that fetches and writes the register needs to identify the saved register block of the debugged program at the time when the program faults. The saved registers can reside either on the stack or a predefined location in memory depending on the platform. When requested by gdb to fetch a register, the stub needs to read the corresponding memory address for the appropriate register. Register writes is analogous to reads. The new register content overwrites the appropriate address in memory before the context of the stopped program is restored.

The memory read and writing part is simpler as the associated address is passed along with the command. The procedure is more difficult without operating system for that the stub may need to consult the page table, if one is required, in order translate the user address into physical address. The sample stub does this through the operating system, which already supports page table lookups and page fault handling. In any case, a stub must implement both register and memory operations.

Finally, a stub must be able to resume a stopped program. It needs to bring back the program context saved in memory and jump to last program counter location of the stopped location.

With only the above-mentioned functionality implemented in a stub, it is an unintelligent piece of code that is driven by the debugger.

In theory, a user can debug a program with a minimal working stub by manually interacting with the stub without the presence of a debugger. However, it is tedious to do so as the user needs to manually send commands to the stub and listen for results from the stub. The analogy here is that the

[2] Stepping a program is a difficult task as the debugger needs to interpret a few instructions of the debugged program.

user writes a large program in assembly instead of c code, which can be readily compiled.

For instance, a debugger provides a user-friendlier interface to help facilitate the speed at which the user can test buggy software. The user improves on efficiency when much of repetitive routines are automated into simple click and dragging. The gdb debugger is the kind of debugger that can builds on top of the minimal stub to provide the full debugging experience.

The debugger, not the stub, is responsible for correlating the correspondence between source code and layout of binary code in memory. In order to do this correspondence, symbol information, which translates between source lines and functions and offsets, are required. The symbol tables basically contain a source line and offset pair, along with other things such as the layout of the variables, etc. For each line of source, the symbol table has an entry that contains the number of bytes beyond the last symbol, which usually is the function entry point. These tables are large in size and may not be feasible to store on the embedded chip. So the debugger, with knowledge of the embedded chip, can store and query the tables instead of the stub.

Symbol table is useful, for example, when the user decides to read the data of a local variable for a stopped program. The debugger consults the symbol table and finds the memory address corresponding to the variable, then issues a memory read command to the stub, and waits for a response.

Another use of the table is when the debugger needs to set a break point. Normally a user does not tell the debugger to stop the program at a certain memory address. The user tells the debugger to stop the program when it reaches a certain line in the source code. The debugger takes this argument and translator it into the memory address for the user. It sets the breakpoint first by recording the content in the memory address through a memory read then writes interrupt instruction to that address.

When a program stops, the user may be interested in knowing where the program stopped. This is accomplished by printing out the stack trace. Stack information, store in registers and memory, is not readily in human readable form. Every chip differs in the way it represents a call stack for its disparate calling conventions, architecture, etc. But all of the stack information, with pointers from registers, are in memory somewhere. The debugger needs to unwind the stack by interpreting it frame by frame. This code can logically exist in either the stub or the debugger. The preferred option is the debugger as the goal is to make the stub as small as possible.

During the frame by frame unwind, the debugger also looks at the return address, which can be in registers or on stack depending on the chip. The debugger consults the symbol table to translate the return address back to the ASCII name of the function. This is done for each frame on the stack so that the stack trace shows functions corresponding to each frame. To identify the topmost frame, the debugger looks at the program counter. The debugger again translates the address stored in the program counter by consulting the symbol table. All of these functionalities are implemented in the debugger and not the stub, and it makes little sense to have around in the stub.

Even with the minimal stub such as the example stub, the debugger can implement other more advanced debugging techniques and tricks. On a sophisticated system, a watch point is set by changing bits in the page table so that access to a particular memory address stops the program by relinquishing control to the stub. The example stub, however, does not accept any commands that deal with page translations. But the stub offers a step command that proves to be useful. A naïve way to implement the watch point is to step the program while interpret the next instruction until there is access to a particular memory address.

Conditional breakpoint is another enhancement. The debugger can easily implement this without addition help from the stub. A conditional breakpoint is a normal breakpoint with addition stopping criterions. The only difference is that the when the debugged program breaks, the debugger evaluates the additional condition, which usually translates into reading and comparing memory contents. The debugger only alerts the user when the evaluation is true; otherwise it continues the program execution.

This concludes the functionalities the need and need not be part of the stub.

Conclusion

Often a debugger cannot run on the embedded chip, but having a small stub that provides some core functionality is a way to mitigate the problem. The size of the stub is limited to the number of function it must provide. To the bare minimum, it must provide functionalities to read write registers and memory, handle exception/signal, and resume a stopped program. On top of these functionalities, the debugger such as gdb can work its magic to provide user a full debugging experience.

Acknowledgements

I thank Professor Stephen Edwards for providing valuable insight on debugging in general, which has gotten me interested in this topic.

References

[1] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. In *The International Journal of High Performance Computing Applications.* 2000.

[2] David R. Hanson and Mukund Raghavachari. A Machine-Independent Debugger. In *Software-Practice And Experience*, Vol. 26(11), 1277-1299 November 1996.

[3] Bill Gatliff. Embedding with GNU: The gdb Remote Serial Protocol. In *Red Hat Developer Network (RHDN)*.

[4] Stan Shebs. An Open Source Debugger for Embedded Development. In *Embedded Systems Conference.* 1999.

[5] Kiyokuni Kawachiya and Takao Moriyama. A Symbolic Debugger for PowerPC-Based Hardware, Using the Engineering Support Processor (ESP). In *IBM Research, Tokyo Research Laboratory.* August, 1997.

[6] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB:The GNU Source-Level Debugger.* Free Software Foundation. January, 2002.